Klassensysteme selbstgebaut

Ein genetischer Weg zu OOP und OOM

von Reinhard Oldenburg

Die Objektorientierung gehört nach wie vor zu den umstrittenen Themen des Informatikunterrichts. In Kortenkamp u.a. (2010) haben wir eine Reihe von Argumenten dazu zusammengetragen und im Ausblick darüber spekuliert, dass ein *genetischer* Weg in die Objektorientierung eine gute Lösung sein könnte.

In der Didaktik unterscheidet man verschiedene Fassungen genetischen Unterrichts. Allen gemeinsam ist, dass sie besonderes Gewicht auf die Entstehung der Konzepte legen. Beim *historisch-genetischen* Unterricht versucht man die tatsächliche historische Entwicklung nachzuzeichnen (vgl. u.a. Oberliesen, 1985). Im Falle der Objektorientierung könnte sich daraus ein interessanter Weg ergeben: Man müsste dann im Kontext von Simulationsprogrammen zur Erkenntnis kommen, dass es günstig ist, bestimmte Daten zusammen mit den relevanten Prozeduren zu kapseln, um so ein zu simulierendes Objekt mit seinen Eigenschaften und seinem Verhalten zu beschreiben. Das ist besonders dann sinnvoll, wenn die Simulation viele gleichartige Objekte umfasst (z.B. Teilchen oder Individuen in einer Schwarmsimulation). Wir werden hier aber einen anderen Weg gehen, der historisch so nicht gegangen wurde, aber hätte gegangen werden können.

Das Ziel dabei ist zu erläutern, wie man von prozeduralen bzw. funktionalen Programmierkonzepten zu denen der Objektorientierung kommt, ohne dass ein Bruch entsteht. Ehlert und Schulte (2007) haben beobachtet, dass der Wechsel von prozeduralem zu objektorientiertem Programmieren eine erhebliche Hürde für die Schülerinnen und Schüler darstellt (vgl. auch Ehlert/Schulte, 2009). Das ist auch nicht verwunderlich, schließlich wird dieser Wechsel oft als Paradigmenwechsel überhöht und durch eine Flut neuer Begriffe begleitet. In diesem Beitrag soll gezeigt werden, dass dies nicht sein muss. Der *graduelle genetische Weg* in die Objektorientierung ist allerdings nur dann gangbar, wenn eine Programmiersprache verwendet wird, die Funktionsabschließungen (Closures) unterstützt. Zum Glück ist das Konzept der Funktionsabschließungen so natürlich, dass man es oft gar nicht bemerkt (man bemerkt eher das Fehlen, wenn Dinge nicht gehen, die ganz natürlich erscheinen), und dieser Artikel wird den Begriff nicht mehr benötigen. JAVA und OBJECTPASCAL scheiden damit aus, geeignet sind PYTHON, RUBY, LUA, BOO, C#, SCHEME, LISP u.v.a. Um konkret und möglichst schulnah zu bleiben, wird durchgängig PYTHON verwendet.

Die hier vorgestellten Ideen sind nicht gänzlich neu. Die Grundideen finden sich schon in dem Klassiker von Abelson und Sussman (1984); es wurde aber versucht, eine Fassung zu finden, die weniger abstrakt ist und insbesondere ohne Lambda-Ausdrücke auskommt. Die unterrichtliche Erprobung dieses Wegs ist geplant; es scheint mir aber sinnvoll, diesen Weg theoretisch zu erörtern, denn er kann auch den Blick von Lehrkräften auf die OOP schärfen.

Überblick über die Konzepte

Grundidee aller hier vorgestellten Konzepte ist, von bekannten Datenstrukturen wie Listen und Dictionaries (assoziativen Reihungen, Hashtabellen) auszugehen und in diesen auch die Funktionen zu ihrer Behandlung zu speichern. Dabei gibt es verschiedene Wege.

Modellieren ist eine offene Tätigkeit; die Ergebnisse können ganz unterschiedlich aussehen, und es stellt sich die Frage nach dem Besser oder Schlechter – natürlich immer in Bezug auf eine Aufgabe, die man erledigen möchte. So haben auch die hier vorgestellten Wege ihre spezifischen Vor- und Nachteile. Gemeinsam ist ihnen, dass jeweils ein eigenes Klassensystem modelliert wird. Das bedeutet, dass die in PYTHON eingebauten OO-Strukturen *nicht* verwendet werden.

Die vier Wege sind unabhängig voneinander. Man kann einen einzigen Weg gehen oder sie auch beliebig kombinieren. In der Darstellung werden wir aber Dinge, die bereits dargestellten Wegen ähneln, nur kurz behandeln, sodass der Aufsatz in der vorgeschlagenen

Reihenfolge gelesen werden sollte. Die Nummerierung der Wege ist willkürlich und kein Hinweis auf das Anspruchsniveau. Die Beispiele sind für die späte Sekundarstufe I oder frühe Sekundarstufe II gedacht.

Lernvoraussetzungen

Voraussetzung für diesen Zugang zur OO ist, dass die Schülerinnen und Schüler mit Listen und Dictionaries (Tabellen) umgehen können. Für beide Datentypen stellt PYTHON Literale zur Verfügung; man kann sie also direkt hinschreiben. Ein Dictionary zur Übersetzung wird etwa so definiert:

```
DE = {"Hund": "dog", "Katze": "cat"}
```

Dann liefert DE["Katze"] wie erhofft *cat*, aber DE["Vogel"] führt zu einem Fehler. Man kann allerdings fragen, ob ein solcher Schlüssel existiert: DE.has_key("Vogel"), und bei Bedarf auch die Tabelle ergänzen: DE["Vogel"] = "bird".

Die zweite zentrale Lernvoraussetzung ist, dass die Schüler Funktionen definieren und diese auch als Objekt, etwa als Parameter für andere Funktionen, benutzen können:

```
A = [34, 2, 3, 5]
def gerade(n):
  return n % 2 == 0
map(gerade, A)
```

Letztes liefert die Liste [True, True, False, False].

Funktionen als Parameter kommen auch bei der Programmierung grafischer Benutzungsoberflächen vor, etwa dann, wenn man so etwas wie button1.setCommand(fun1) hinschreibt, wobei fun1 die mit def definierte Funktion ist, die beim Drücken des Knopfes aufgerufen werden soll.

Benötigt werden im Folgenden noch zwei speziellere Fähigkeiten, nämlich die Definition von Funktionen mit beliebig vielen Parametern. In PYTHON geht das so:

```
def test(a, b, *rest):
    print a, b, rest
```

Dann liefert test(1, 2, 3, 4) die Ausgabe *1, 2, (3, 4)*. Die restlichen Argumente wurden als Tupel gefasst. Man kann dieses Tupel durch L = list(rest) in eine Liste verwandeln.

Ein letzter wichtiger Punkt ist der Befehl apply, der eine Funktion auf Daten anwendet, die in einer Liste oder einem Tupel vorliegen. Angenommen, man hat eine Funktion zur Mittelwertsberechnung zweier Zahlen definiert:

```
def mittelwert(a, b): return (a + b)/2.0
```

Wenn die Daten aber in einer Liste vorliegen, wie beispielsweise L = [4, 6], kann man nicht direkt mittelwert(L) schreiben, weil die Funktion ja zwei Zahlen als Eingabe erwartet, nicht aber eine Liste. Das Problem löst man mit apply(mittelwert, L). Allgemein ist apply(f, $[x_1, \ldots, x_n]$) äquivalent zu $f(x_1, \ldots, x_n)$.

Weg 1: Listen mit „Methoden“ und „Klassenköpfen“

Die Datenstruktur *Liste* ist vielfältig nutzbar: Man kann allerlei Objekte einfügen und so zusammen speichern, sie an Funktionen übergeben und so weiter. Dabei kommt es allerdings häufig vor, dass Daten gleich aussehen, die unterschiedliche Bedeutung haben. Eine Liste mit drei Zahlen wie [255, 100, 0] könnte einen RGB-Farbwert darstellen, aber genauso gut einen Punkt im 3-D-Raum. Wenn man beides in dieser Form mit Listen beschreibt, muss man als Programmierer stets bedenken, wofür die Zahlen stehen sollen. Insbesondere kann das Programm selbst nicht merken, ob eine Verwechslung vorliegt. Eventuell kommt es dann zu einem Laufzeitfehler, weil ein Farbwert als Raumpunkt interpretiert wurde, oder umgekehrt irritiert ein negativer Wert für einen Farbkanal die Grafikbibliothek.

Um solche Verwechslungen zu vermeiden, ist es sinnvoll, die Listen mit jeweils einem „Kopf“ zu versehen, der über die Art der Daten Auskunft gibt. Dazu schreibt man etwa:

```
a = ["Punkt", 14.5, 1.2, 20]
b = ["Punkt", -4, 2, 3]
f = ["Farbe", 10, 10, 255]
g = ["Farbe", 255, 255, 255]

def typ(a): return a[0]
```

Der Betrag eines Vektors ist durch den „räumlichen Pythagoras“ definiert. Als Betrag einer Farbe nimmt man am besten die Summe der Helligkeiten.

```
def Betrag(A):
  if typ(A) == "Punkt":
    return math.sqrt(A[1]**2 + A[2]**2 + A[3]**2)
  if typ(A) == "Farbe": return A[1] + A[2] + A[3]
  print "Fehler: Falsche Datentypen in Betrag"

print "Betrag von a = ", Betrag(a)
print "Betrag von f = ", Betrag(f)
```

Farben dürfen keine negativen Komponenten haben. Am besten prüft man das gleich bei der Erstellung von Farben. Man führt deshalb *Konstruktoren* ein; das sind ganz normale Funktionen, die eine solche Liste zusammenstellen und gleich die nötigen Prüfungen vornehmen:

```
def makePunkt(x, y, z): return ["Punkt", x, y, z]
def makeFarbe(r, g, b):
  if r < 0 or g < 0 or b < 0 or r > 255 or g
                                  > 255 or b > 255:
    raise Exception("Falsche Farbwerte")
return ["Farbe", int(r), int(g), int(b)]
```

Damit lassen sich Punkte und Farben bequem erzeugen. Wenn man anfängt, neben dem Betrag noch weitere Methoden zu implementieren, stellt man schnell fest, dass es sehr ungünstig ist, wenn das Wissen über Farben und das über Punkte sich über die ganze Datei verteilt. Außerdem ist es umständlich, ständig Typabfragen zu programmieren. Die rettende Idee ist, die Funktionen auch mit in die Listen zu packen. Dazu muss man den Konstruktor modifizieren:

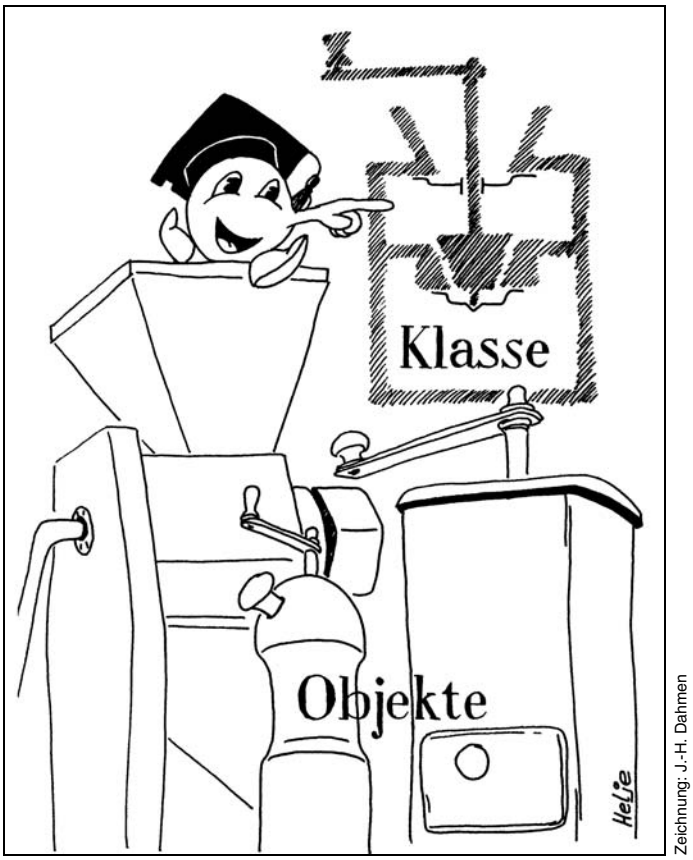
```
def makePunkt(x, y, z):
  A = ["Punkt", x, y ,z]
  def Betrag(A):
    return math.sqrt(A[1]**2 + A[2]**2 + A[3]**2)
  def Mittelpunkt(A, B):
    return makePunkt((A[1] + B[1])/2.0,
                     (A[2] + B[2])/2.0,
                     (A[3] + B[3])/2.0)
  def Beschreibe(A):
    return "[" + str(A[1]) + "," + str(A[2])
               + "," + str(A[3]) + "]"
  methodenliste = ["Betrag", Betrag,
                   "Mittelpunkt", Mittelpunkt,
                   "Beschreibe", Beschreibe]
  return A + [methodenliste]
```

Wenn man jetzt z. B. mit a = makePunkt(1, 2, 3) einen Punkt erzeugt, kann man auf ihn eine seiner „eigenen“ Methoden anwenden. Man muss sie dazu aus der Methodenliste (dem letzen Eintrag der Gesamtliste) herausholen und anwenden:

```
>>> P = makePunkt(7,3,1)
>>> P[1]
7
>>> P[-1] # Das liefert die Methodenliste
['Betrag', <function Betrag at 0x02C9B230>,
           'Mittelpunkt', <function Mittelpunkt
            at 0x02C9B1F0>, 'Beschreibe',
           <function Beschreibe at 0x02C9B270>]
>>> P[-1][1]
<function Betrag at 0x02C9B230>
```

Jetzt kommt der Aufruf der Methode auf P:

```
>>> P[-1][1](P)
7.6811457478686078
```



Zeichnung: J.-H. Dahmen

Das macht leider ziemlich viel Arbeit. Man benötigt eine Funktion, die aus der Methodenliste die passende Methode herausholt und anwendet. Das geht wie folgt:

```
def call(objekt,methode,*args):
  alleargs = [objekt] + list(args)
             # das ist [objekt, arg1,..]
  methodenliste = objekt[-1] # Liste der Methoden
  for i in range(len(methodenliste)):
    if methodenliste[i] == methode:
      return apply(methodenliste[i+1], alleargs)
  print "Methode unbekannt"
```

Aufgerufen wird das Ganze etwa so:

```
a = makePunkt(1,2,3)
b = makePunkt(2,0,7)
print "Beschreibe a ", call(a, "Beschreibe")
print "Beschreibe b ", call(b, "Beschreibe")
c = call(a, "Mittelpunkt", b)
print "Mittelpunkt c von a und b",
              call(c, "Beschreibe")
print "Betrag von c ", call(c, "Betrag")
```

In der Datei Weg1OOListen.py (siehe LOG-IN-Service) ist außerdem ein Konstruktor für Elemente einer Farbklasse enthalten.

Der Wunsch nach Vererbung entsteht, wenn man Objekte beschreiben will, die bereits vorhandenen Objekten ähneln. Das ist etwa der Fall, wenn man in einem Geometrieprogramm Punkte verwendet. Diese sind zum einen natürlich Punkte mit ihren Koordinaten, haben aber auch einen Namen und eine Farbe. Man muss also die Liste um die neuen Attribute erweitern und auch die Methodenliste entsprechend erweitern oder ändern. Man könnte hier etwa auch Methoden herausnehmen, sodass sie in der Unterklasse nicht mehr verfügbar sind (was in JAVA z. B. nicht geht, wohl aber in EIFFEL).

```
def makeGeoPunkt(name,x,y,z,r,g,b):
  A = makePunkt(x,y,z)
            # Das ist der Akt der Vererbung
  A[0] = "GeoPunkt"
            # Die neue Klasse heißt aber anders
  methodenliste = A[-1]
            # Methodenliste getrennt behandeln
  del A[-1] # Alte Methodenliste wegnehmen
  A.append(makeFarbe(r,g,b))
            # Neue Attribute anfügen
  A.append(name)
  def Beschreibe(A):
    return "GeoPunkt " + A[5] + "[" + str(A[1])
                       + "," + str(A[2])
                       + "," + " " + str(A[3])
                       + "] Farbe = " + call(A[4],
                       "Beschreibe")
  def setzeNamen(A, nam): A[5] = nam # Neue Methode
  neuemethodenliste = ["Beschreibe", Beschreibe,
                       "setzeNamen", setzeNamen]
  return A + [neuemethodenliste + methodenliste]

gp = makeGeoPunkt("Q",1,2,3,255,0,0)
print
print call(gp, "Beschreibe")
call(gp, "setzeNamen", "Q1")
print call(gp, "Beschreibe")
```

Damit ist alles beisammen, was ein Klassensystem ausmacht. In der Datei Weg1OOPythonKlassen.py findet man dieses Beispiel mit dem Klassensystem von PYTHON realisiert und hat somit einen „Stein von Rosetta“ für die Übersetzung. Die wesentlichen Unterschiede zwischen

Tabelle 1.

Operation	PYTHON	Selbstgebaut
Attribut eines Objekts	Obj.attr	Obj[n] # n ist die Nummer des Attributs
Methodenaufruf	Obj.methode(args, …)	call(Obj, "methode", args, …)

der PYTHON-Syntax und unserer wird in Tabelle 1 zusammengefasst.

Weg 2: Methoden in Dictionaries

In einem Geometrieprogramm sollen Punkte, Geraden usw. verarbeitet werden. Damit man mit einem Punkt als Objekt umgehen kann, sollte man alle Informationen, die einen Punkt betreffen, in einer Datenstruktur speichern. Natürlich kann man hier gleich von den Attributen eines Punktes sprechen. Es bieten sich Listen oder Tabellen (Dictionaries) an. Letztere haben den Vorteil, dass man die einzelnen Attribute unter verständlichen Namen statt unter der Position in einer Liste erreicht.

Da man in einem Programm sehr viele Punkte erzeugen wird, bietet es sich an, das Herstellen der Tabelle durch eine Funktion erledigen zu lassen. Da sowohl Punkte als auch andere Objekte als Tabellen gespeichert werden, ist es sinnvoll, hinzuzufügen, dass es sich bei einer bestimmten Tabelle tatsächlich um einen Punkt handelt. Ein erster Anlauf für eine erzeugende Funktion, die man im Folgenden *Konstruktor* nennt, könnte sein:

```
def mkPunkt(name, x, y):
  punkt = {"x":x, "y":y, "name":name,
           "Klasse":"Punkt"}
return punkt
```

Jetzt wird man Funktionen definieren, die mit diesen Daten arbeiten. Beispielsweise kann es sinnvoll sein, einen Punkt nach rechts verschieben zu können:

```
def nachRechts(P, dx):
  P["x"] = P["x"] + dx
```

Allerdings wird man auch Kreise, Strecken und Geraden verschieben können wollen. Was sind mögliche Vorgehensweisen?

▷ Man definiert verschiedene Funktionen PunktNachRechts, KreisNachRechts.

▷ Man führt eine Fallunterscheidung ein:

```
def nachRechts(Objekt, dx):
  if Objekt["Klasse"] == "Punkt":
     Objekt["x"] = Objekt["x"] + dx
  if ["Klasse"] == "Gerade":
     Objekt["a"] = ...
```

Der erste Weg hat den Nachteil, dass es sehr mühsam wird, alle Objekte der Konstruktion gemeinsam zu verschieben; man muss nämlich immer die Klasse ermitteln und die richtige Funktion aufrufen. Der zweite Weg hat den Nachteil, dass beim Einfügen weiterer Objektklassen alle Funktionen durchforstet werden müssen, um die Änderungen anzubringen.

Die Lösung dieser Probleme ist eigentlich einfach, wurde aber in der Geschichte der Informatik nicht schnell gefunden und benötigte lange Zeit, um sich durchzusetzen. Es ist daher nicht zu erwarten, dass Schüler oder Schülerinnen von alleine darauf kommen. Die Idee ist, dass man die Funktionen, die sich auf eine Klasse beziehen, im Konstruktor mitdefiniert und in einer Tabelle (der *Methodentabelle*) sammelt:

```
def mkPunkt(name, x, y):
  punkt = {"x":x, "y":y, "name":name,
           "Klasse":"Punkt"}
  def setzeName(ich, neu):
    ich["name"] = neu
  def betrag(ich): return sqrt(ich["x"]**2
                   + ich["y"]**2)
  def nachRechts(ich, dx):
    ich["x"] = ich["x"] + dx
  def beschreibe(mich):
    return mich["Klasse"] + ":" + mich["name"] + ",
      x = " + str(mich["x"]) + ", y = " +
str(mich["y"])
methoden = {"setzeName": setzeName,
            "betrag":betrag,
            "nachRechts":nachRechts,
            "beschreibe":beschreibe}
punkt["methoden"] = methoden
return punkt
```

Um die Methoden aufzurufen, hat man jetzt aus der Methodentabelle die entsprechende Methode aufzurufen, etwa so:

```
>>> P = mkPunkt("P", 4, 7)
>>> beschreibeMethode = P["methoden"]["beschreibe"]
>>> beschreibeMethode(P)
'Punkt: P, x = 4, y = 7'
```

Das ist umständlich und sollte von einer Funktion erledigt werden:

```
def call(obj, methodenname, *args):
  alleArgs = [obj] + list(args)
             # Das ist [obj,arg1,arg2,..]
  methodenTabelle = obj["methoden"]
             # Methoden-Tabelle
  methode = methodenTabelle[methodenname]
             # Methoden-Funktion
  return apply(methode, alleArgs)
```

Damit können Punkte erzeugt und Methoden aufgerufen werden:

Tabelle 2.

Operation	PYTHON	Selbstgebaut
Attribut eines Objekts	Obj.attr	Obj["attr"]
Methodenaufruf	Obj.methode(args, …)	call(Obj, "methode", args, …)

```
p1 = mkPunkt("P1", 5, 6)
p2 = mkPunkt("P2", 5, 9)
print call(p1, "beschreibe")
print call(p2, "beschreibe")
call(p1, "nachRechts",50)
print call(p1, "beschreibe")
call(p1,"setzeName", "P1neu")
print call(p1, "beschreibe")
print call(p2, "beschreibe")
```

An dieser Stelle kann man auch einen Vergleich mit dem eingebauten Klassensystem von PYTHON anstellen. Man erhält eine ähnliche Tabelle wie bei Weg 1 (siehe Tabelle 2, vorige Seite).

Natürlich ist das vordefinierte System eleganter und besitzt eine knappere und zudem mit vielen anderen objektorientierten Sprachen gemeinsame Syntax. Der Vorteil unseres Zugangs ist aber, dass man alles im Griff hat und z. B. auch verstehen kann, wozu der Parameter self in allen Methoden angegeben werden muss, die man mit dem in PYTHON eingebauten Klassensystem schreibt. Methoden sind nämlich im Grunde ganz normale Funktionen, die wissen müssen, auf welches Objekt sie angewendet werden sollen.

Das Ableiten einer Klasse von einer anderen ist jetzt ganz offensichtlich; man könnte es Schülerinnen und Schülern als Problemlösungsaufgabe stellen. Zunächst modelliert man auf abstrakter Ebene, welche weiteren Attribute und Methoden ein Mittelpunkt als spezieller Punkt benötigt; dann fragt man, ob man einen Konstruktor für Mittelpunkte so definieren kann, dass man die bereits vorhandene Funktionalität weitgehend weiterverwenden kann (und natürlich soll man nicht Programmtext kopieren, sodass auch Mittelpunkte von Verbesserungen bei Punkten automatisch profitieren). Eine Lösung könnte so aussehen:

```
def mkMittelpunkt(name, P, Q): # P Q sind Punkte
  punkt = mkPunkt(name, 0, 0)
  punkt["Klasse"] = "Mittelpunkt"
  punkt["P"] = P # Neue Attribute
  punkt["Q"] = Q
  def berechne(mich): # Neue Methoden
    mich["x"] = (P["x"] + Q["x"])/2.0
    mich["y"] = (P["y"] + Q["y"])/2.0
berechne(punkt)
punkt["methoden"]["berechne"] = berechne
return punkt

M = mkMittelpunkt("M1", p1, p2)
print
print call(M, "beschreibe")
call(p1, "nachRechts", 50)
call(p2, "nachRechts", 50)
print call(p1, "beschreibe")
print call(p2, "beschreibe")
call(M, "berechne")
print call(M, "beschreibe")
```

Alles zusammen findet man in der Datei Weg2OODict.py (siehe LOG-IN-Service).

Dieser zweite Weg wird verwendet, um ein dynamisches Geometrieprogramm zu schreiben (siehe LOG-IN-Service, S. 143). Das zeigt, dass es sich nicht um ein „Wegwerfprogramm“ handelt. Ein Unterschied und Nachteil zum richtigen Klassensystem sei aber nicht verschwiegen: Es wird (übrigens auch schon beim ersten, nicht aber beim dritten Weg) viel Speicherplatz verschwendet, weil die Methoden nicht nur einmal, sondern bei jedem Objekt gespeichert werden. Das lässt sich natürlich besser machen (siehe die Datei Weg2OODictKlassen.py), es ist aber eine offene Frage, ob man für das prinzipielle Verständnis dies tun muss. Andererseits eröffnet man sich damit auch den Weg zu Klassenattributen und Klassenmethoden.

Weg 3: Prototypische Objektorientierung

Beim Herstellen realer Objekte gibt es zum einen den Fall, dass der komplette Bauplan vorab ausgedacht wird und dann die Objekte diese Art (Klasse) danach produziert werden. Dies entspricht dem Mehrheitsweg der OO. Es gibt aber auch einige wenige andere Sprachen, in denen es gar keine Klassen gibt und die doch objektorientiert sind (beispielsweise die Sprache SELF). In diesen Sprachen geht man vor wie ein Tüftler: Statt vorab den kompletten Bauplan zu entwerfen, bastelt man an einem Prototypen, und wenn man mit dem zufrieden ist, geht man in die Produktion, indem man ihn kopiert. In diesem Stil wiederholen wir jetzt das Beispiel einer selbstgebauten Punktklasse. Zunächst wird ein Prototyp definiert:

```
Punkt = {}
Punkt["x"] = 5
Punkt["y"] = 10
Punkt["name"] = "P1"
Punkt["ist"] = "Punkt"
```

Mit den ersten Zeilen modelliert man die Attribute. Zu jedem Zeitpunkt kann man mit dem Objekt schon etwas Vernünftiges anfangen. Beispielsweise kann man ein paar nützliche Funktionen (Methoden) definieren und am Prototypen ausprobieren:

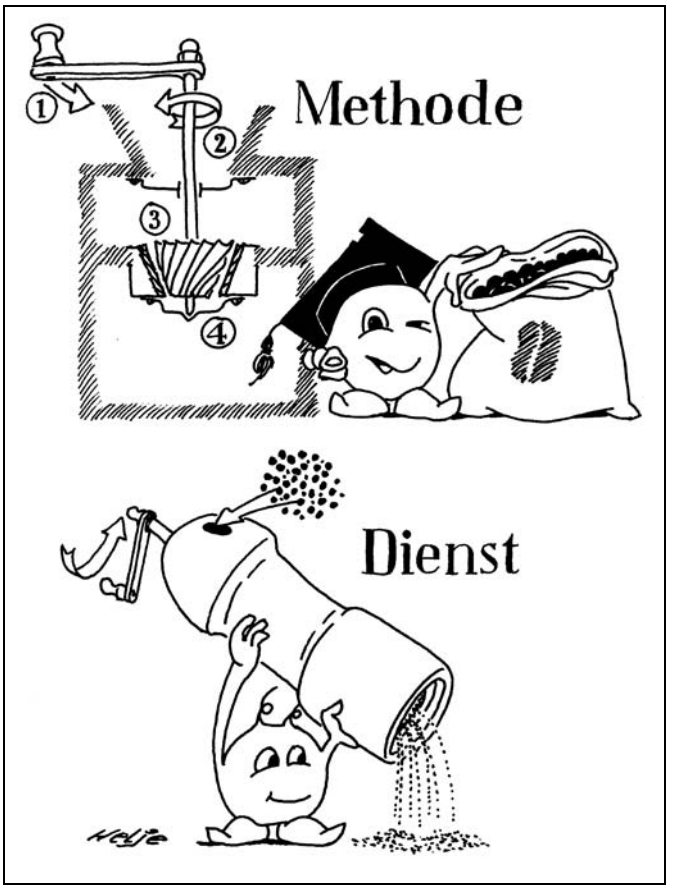
```
def abstandVon(ich, du):
  return math.sqrt((ich["x"] - du["x"])**2 +
                   (ich["y"] - du["y"])**2)
def alsStr(ich): return
  "[" + str(ich["x"]) + "|" + str(ich["y"]) + "]"

def verschiebe(ich, dx, dy):
  ich["x"] += dx
  ich["y"] += dy
```

Wenn man damit soweit zufrieden ist, kann man in die Serienproduktion gehen. In PYTHON muss man dazu die Bibliothek copy importieren. Dann erschafft man die neuen Objekte durch eine tiefe Kopie (deepcopy, in anderen Sprachen könnte sie auch clone heißen):

```
A = copy.deepcopy(Punkt)
B = copy.deepcopy(Punkt)
A["x"] = 9
A["y"] = 1
[B["x"], B["y"]] = [3,3]
         # Koordinaten setzen als Einzeiler
```

Damit lässt sich schon ganz gut arbeiten. Aber man hat noch keine Zuordnung der Methoden: Wenn mehrere „Prototypenklassen“ definiert sind, hat man selbst dafür zu sorgen, immer die richtigen anzuwenden. Das



Zeichnung: J.-H. Dahmen

geht natürlich besser, und daher hat schon der Prototyp die Information bekommen, dass er ein „Punkt“ ist. Also packen wir jetzt die Methoden für Punkte zusammen und definieren eine Funktion call, die zu einem Objekt die zu seiner Klasse gehörige Methode aufruft:

```
methoden = {}
methoden["Punkt"] = {"abstandVon":abstandVon,
                     "alsStr": alsStr,
                     "verschiebe": verschiebe}

def call(objekt, methodenName, *args):
  global methoden
  klassenName = objekt["ist"]
  methodenListe = methoden[klassenName]
  methode = methodenListe[methodenName]
  return apply(methode,[objekt] + list(args))

print "A = ", call(A, "alsStr")
print "B = ", call(B, "alsStr")
print "Abstand A, B = ", call(A, "abstandVon", B)
print "Jetzte Verschiebung von A um 10, 11"
call(A,"verschiebe", 10, 11)
print "A = ",call(A, "alsStr")
print "Abstand A, B = ",call(A, "abstandVon",B)
```

Das tut, was man erwartet. Natürlich kann man auch einen erfolgreichen Prototypen gründlich umbauen, sprich *Vererbung* einführen. Ein Kreis etwa ist ein Punkt, der einen Radius hat:

```
Kreis = copy.deepcopy(Punkt)
Kreis["r"] = 5
Kreis["ist"] = "Kreis"
methoden["Kreis"] = methoden["Punkt"]
def KreisalsStr(ich):
  return
"Kreis[" + str(ich["x"]) + "|" + str(ich["y"])+ ";
  radius = "+ str(ich["r"])+"]"
methoden["Kreis"]["alsStr"] = KreisalsStr

K1 = copy.deepcopy(Kreis)
K1["r"] = 10
print "K1 = ", call(K1, "alsStr")
```

Mehrfachvererbung ist – wie auch im richtigen Leben – knifflig, aber möglich. Dazu muss man eine der Elternklassen, also ihren Prototypen, auswählen, ihn kopieren und die Felder noch hinzufügen, die in den anderen Eltern vorkommen. Ebenso müssen die Methodentabellen zusammengefügt werden.

Weg 4: Objektorientierung mit Geheimnissen

Bei den ersten drei Wegen sind die Objekte klassische Datenstrukturen; sie konnten daher sukzessive entwickelt werden. Ein Nachteil ist aber, dass das Geheimnisprinzip verletzt wird: Wer Zugriff auf ein Objekt hat, kann alle Attribute lesen und ändern. Das lässt sich nur vermeiden, wenn man die Informationen als lokale Variablen versteckt, auf die von außerhalb nicht zugegriffen werden kann. Das Objekt selbst ist dann eine Funktion im Sinne von PYTHON (oder einem „sauberen“ Block in SMALLTALK)!

Um zu einer eigenen Klasse zu gelangen, definieren wir – wie schon in den anderen Wegen – einen Konstruktor, also eine Funktion, die ein Objekt erstellt. Dieses Objekt ist jetzt aber keine Liste oder eine der üblichen Datenstrukturen, sondern eine Funktion, die zurückgegeben wird und mit deren Aufruf man die verschiedenen Methoden ansteuert. Eine Methodentabelle wie in den anderen Wegen gibt es nicht explizit, sondern implizit in der Dispatch-Funktion call, die als „Objekt“ vom Konstruktor zurückgegeben wird.

Das Programm wird dadurch recht kompakt, lässt sich aber nicht so bruchlos sukzessive aus „klassischen“ Datenstrukturen entwickeln, wie das bei den anderen Wegen der Fall ist:

```
def mkPunkt(x, y):
  this = {"x":x, "y":y, "name":"?"}
  def setName(this, n): this["name"] = n
  def getName(this): return this["name"]
  def getX(): return this["x"]
  def getY(this): return this["y"]
  def setX(this,x): this["x"] = x
  def setY(this,y): this["y"] = y
  def move(this, dx, dy):
    this["x"] += dx
    this["y"] += dy

def call(meth, *args):
  if meth == "setName":
              return setName(this, args[0])
  if meth == "getX": return getX()
  if meth == "getY": return getY(this)
  if meth == "setX": return setX(this, args[0])
  if meth == "setY": return setY(this, args[0])
  if meth == "move": return move(this,
                      args[0], args[1])
```

```
return call

P = mkPunkt(3, 8)
P("setName", "P")
print "P: x = ", P("getX"), " y = ", P("getY")
Q = mkPunkt(1, 1)
Q("setName", "Q1")
print "Q: x = ", Q("getX"), " y = ", Q("getY")
print "Jetzt P verschieben um [9, -5]!"
P("move", 9, -5)
print "P: x = ", P("getX"), " y = ", P("getY")
```

Anders als in den ersten drei Wegen kann in diesem Weg nicht auf die Attribute direkt zugegriffen werden: Jeder Zugriff muss über get/set-Methoden laufen.

Im Gegensatz zu den anderen Verfahren ist der Methodenaufruf auch ohne eine call-Funktion erträglich einfach: Statt dem OO-üblichen Format obj.meth(arg1, …) hat man hier, wie die Beispiele zeigen, obj("methname", arg1, …) zu schreiben.

Fazit

Mit den Beispielen dieses Beitrags sollte gezeigt werden, wie man Klassensysteme selbst bauen kann. Für engagierte Schülerinnen und Schüler, die sich ungern den Entscheidungen von Sprachentwicklern ausliefern, ist damit ein Experimentierfeld gegeben, auf dem vielerlei Dinge möglich sind. Man kann Strategien der Mehrfachvererbung ausprobieren, oder man definiert neben Methoden, die zu einer Klasse gehören, Methoden, die als Brückenelemente zwischen zwei Klassen dienen. Dann könnte man statt toString, toInteger und toFloat eine einzige Methode to oder convert verwenden, die – je nach den Klassen der Argumente – das Gewünschte bewirkt.

Die Hauptperspektive des Artikels ist aber eine andere: Es sollte gezeigt werden, dass die Konzepte der Objektorientierung kein grundlegend neues Paradigma darstellen, sondern sich ganz natürlich aus dem Bemühen ergeben, Daten und Prozeduren zweckmäßig zu strukturieren. Damit ist ein genetischer Weg in die OOP möglich. Unterrichtserfahrungen dazu stehen allerdings noch aus.

Prof. Dr. Reinhard Oldenburg
Institut für Didaktik
der Mathematik und der Informatik
Goethe-Universität
Senckenberganlage 9
60325 Frankfurt

E-Mail: oldenburg@math.uni-frankfurt.de

Im **LOG-IN-Service** (siehe Seite 143) stehen alle PYTHON-Programme des Beitrags sowie zwei weitere Beispiele („Entwicklung eines Dynamischen Geometrieprogramms“ und „Rekursion aus Klassen“ als PDF-Dateien) zum Herunterladen zur Verfügung.

Literatur

Abelson, H.; Sussman, G. J.; Sussman, J.: Struktur und Interpretation von Computerprogrammen. Berlin: Springer, 1991.

Börstler, J.; Hadar, I.: Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts – Report on the 11th Workshop TLOOC at ECOOP 2007. In: M. Cebulla (Hrsg.): Object-Oriented Technology – ECOOP 2007 Workshop Reader, Berlin, Germany, July 2007, Final Reports. Reihe „Lecture Notes in Computer Science“, Band 4906. Berlin u. a.: Springer, 2008, S. 182–192.

Ehlert, A.; Schulte, C.: Learners Views on Objects-First and Objects-Later – Results of an Exploratory Study. Berlin: Manuskript, 2007 (siehe: Börstler/Hadar, 2008, S. 184 und 185).

Ehlert, A.; Schulte, C.: Unterschiede im Lernerfolg von Schülerinnen und Schülern in Abhängigkeit von der zeitlichen Reihenfolge der Themen (OOP-First bzw. OOP-Later). In: B. Koerber (Hrsg.): Zukunft braucht Herkunft – 25 Jahre „INFOS – Informatik und Schule“. INFOS 2009 – 13. GI-Fachtagung Informatik und Schule, 21.–24. September 2009 in Berlin. Reihe „GI-Edition Lecture Notes in Informatics“, Band P-156. Bonn: Köllen Verlag, 2009, S. 121–132.

Kortenkamp, U.; Modrow, E.; Poloczek, J.; Oldenburg, R.; Rabel, M.: Objektorientierte Modellierung – aber wann und wie? Zur Bedeutung der OOM im Informatikunterricht. In: LOG IN, 29. Jg. (2009), H. 160/161, S. 41–47.

Oberliesen, R.: Informationstechnologische Bildung und historisch-genetisches Lernen. In: LOG IN, 5. Jg. (1985), H. 4, S. 25–30.